**Abstract Function:**

An abstract function or abstract method in an OOP language is a function or method used to override the behavior of the function in an inherited class with the same signature to achieve the polymorphism.

**Abstract Classes:**

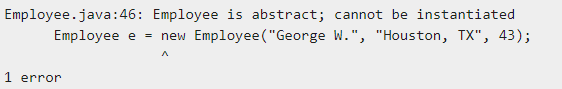
|  |
| --- |
| **Example** |
| abstract class Employee  {  private String name;  private String address;  private int number;  public Employee(String name, String address, int number) {  System.out.println("Constructing an Employee");  this.name = name;  this.address = address;  this.number = number;  }  public double computePay() {  System.out.println("Inside Employee computePay");  return 0.0;  }  public void mailCheck() {  System.out.println("Mailing a check to " + this.name + " " + this.address);  }  public String toString() {  return name + " " + address + " " + number;  }  public String getName() {  return name;  }  public String getAddress() {  return address;  }  public void setAddress(String newAddress) {  address = newAddress;  }  public intgetNumber() {  return number; } } |

You can observe that except abstract the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way

|  |
| --- |
| public class AbstractDemo {  public static void main(String [] args) {  /\* Following is not allowed and would raise error \*/  Employee e = new Employee("George W.", "Houston, TX", 43);  System.out.println("\n Call mailCheck using Employee reference--");  e.mailCheck();  } } |

When you compile the above class, it gives you the following error



Inheriting the Abstract Class:

We can inherit the properties of Employee class just like concrete class in the following way

|  |
| --- |
| public class Salary extends Employee {  private double salary; // Annual salary  public Salary(String name, String address, int number, double salary) {  super(name, address, number);  setSalary(salary);  }  public void mailCheck() {  System.out.println("Within mailCheck of Salary class ");  System.out.println("Mailing check to " + getName() + " with salary " + salary);  }  public double getSalary() {  return salary;  }  public void setSalary(double newSalary) {  if(newSalary>= 0.0) {  salary = newSalary;  }  }  public double computePay() {  System.out.println("Computing salary pay for " + getName());  return salary/52;  } } |

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

|  |
| --- |
| public class AbstractDemo {  public static void main(String [] args) {  Salary s = new Salary("MohdMohtashim", "Ambehta, UP", 3, 3600.00);  Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);  System.out.println("Call mailCheck using Salary reference --");  s.mailCheck();  System.out.println("\n Call mailCheck using Employee reference--");  e.mailCheck();  } } |

## Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

* **abstract** keyword is used to declare the method as abstract.
* You have to place the **abstract** keyword before the method name in the method declaration.
* An abstract method contains a method signature, but no method body.
* Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Following is an example of the abstract method.

|  |
| --- |
| public abstract class Employee {private String name;private String address;private int number;public abstract double computePay();// Remainder of class definition} |

Declaring a method as abstract has two consequences −

* The class containing it must be declared as abstract.
* Any class inheriting the current class must either override the abstract method or declare itself as abstract.

**Note** − Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below

|  |
| --- |
| public class Salary extends Employee {private double salary; // Annual salarypublic double computePay() {System.out.println("Computing salary pay for " + getName());return salary/52;}// Remainder of class definition} |

**Polymorphism:**

Polymorphism is a generic term that means 'many shapes'. More precisely Polymorphism means the ability to request the same operations be performed by a wide range of different types of things. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

|  |
| --- |
| package javaapplication4;  abstract class shape  {  abstract void display();  }  class square extends shape  {  void display()  {  System.out.println("Square");  }  }  class circle extends shape  {  void display()  {  System.out.println("Circle");  }  }  public class JavaApplication4  {  public static void main(String[] args)  {  shape s1= new circle();  s1.display();  s1=new square();  s1.display();  }  } |

**Interface in Java:**

An interface in Java is a blueprint of a class. It has abstract methods and variables.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritances in Java.

In other words, you can say that interfaces can have abstract methods. It cannot have a method body.

|  |
| --- |
| 1. **interface** <interface\_name>{ 3. // declare constant fields 4. // declare methods that abstract 5. // by default. 6. } |

**Example 1:**

|  |
| --- |
| package javaapplication5;   1. **interface** printable{ 2. **void** print(); 3. } 4. **class** A6 **implements** printable 5. { 6. **public** **void** print() 7. { 8. System.out.println("Hello"); 9. }   }  **Public class** JavaApplication5  {  **public** **static** **void** main(String args[])  {  A6 obj=new A6():  Obj.print();   }  } |

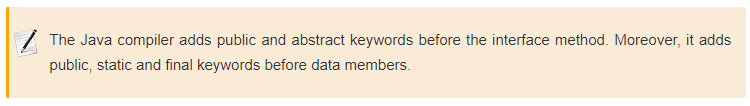
**Example 2:**

|  |
| --- |
| package javaapplication5;  interface Bank  {  float rateOfInterest();  }  class SBI implements Bank  {  public float rateOfInterest()  {  return 9.15f;  }  }  class PNB implements Bank  {  public float rateOfInterest()  {  return 9.7f;  }  }  public class JavaApplication5  {  public static void main(String[] args)  {  Bank b=new SBI();  System.out.println("ROI: "+b.rateOfInterest());  }  } |

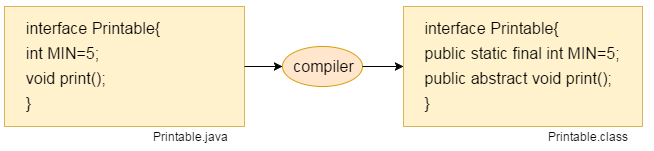
**Interface Improvement:**

Since [Java 8](https://www.javatpoint.com/java-8-features), interface can have default and static methods.

Internal addition by the compiler:



In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



Interface variables are static **because java interfaces cannot be instantiated on their own**. The value of the variable must be assigned in a static context in which no instance exists. The final modifier ensures the value assigned to the interface variable is a true constant that cannot be re-assigned.

**Example 3:**

|  |
| --- |
| interface Printable  {  int MIN=5;  void print();  }  class A6 implements Printable  {  public void print()  {  // MIN=7; cannot assign a value to final variable MIN  System.out.println(MIN);  }  }  class HelloWorld {  public static void main(String[] args)  {  System.out.println(Printable.MIN);  A6 obj = new A6();  obj.print();  }  } |

Default Method in Interface:

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

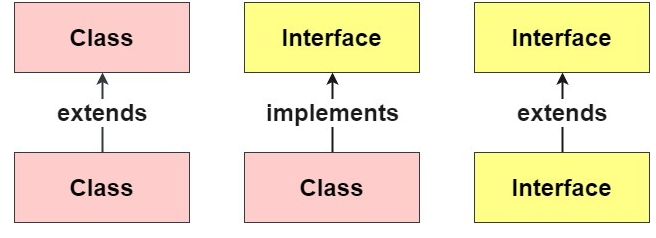
|  |
| --- |
| 1. **interface** Drawable{ 2. **void** draw(); 3. **default** **void** msg(){System.out.println("default method");} 4. } 5. **class** Rectangle **implements** Drawable{ 6. **public** **void** draw(){System.out.println("drawing rectangle");} 7. } 8. **class** TestInterfaceDefault{ 9. **public** **static** **void** main(String args[]){ 10. Drawable d=**new** Rectangle(); 11. d.draw(); 12. d.msg(); 13. }} |

Static Method in Interface:

|  |
| --- |
| 1. **interface** Drawable{ 2. **void** draw(); 3. **static** **int** cube(**int** x){**return** x\*x\*x;} 4. } 5. **class** Rectangle **implements** Drawable{ 6. **public** **void** draw(){System.out.println("drawing rectangle");} 7. } 9. **class** TestInterfaceStatic{ 10. **public** **static** **void** main(String args[]){ 11. Drawable d=**new** Rectangle(); 12. d.draw(); 13. System.out.println(Drawable.cube(3)); 14. }} |

**Relation between Class and Interface:**

As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.



**Multiple inheritance in JAVA (through interface):**

|  |
| --- |
| package javaapplication5;   1. **interface** Printable{ 2. **void** print(); 3. } 4. **interface** Showable{ 5. **void** show(); 6. } 7. **class** A7 **implements** Printable,Showable 8. { 9. **public** **void** print(){System.out.println("Hello");} 10. **public** **void** show(){System.out.println("Welcome");} 11. }   public class JavaApplication5 {   1. **public** **static** **void** main(String args[]){ 2. A7 obj = **new** A7(); 3. obj.print(); 4. obj.show(); 5. } 6. } |

**Interface Inheritance:**

|  |
| --- |
| interface Printable{  void print();  }  interface Showable extends Printable{  void show();  }  class A7 implements Showable  {  public void print(){System.out.println("Hello \n");}  public void show(){System.out.println("Welcome");}  }  Public class HelloWorld {  public static void main(String[] args)  {  A7 obj = new A7();  obj.print();  obj.show();  }  } |